Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA ILLINOIS 61801

CAC Document Number 239
CCTC-WAD Document Number 7517

Networking Research in Front Ending
and Intelligent Terminals

Unix/ENFE Experimental Performance Report

September 30, 1977

CAC Document Number 239
CCTC-WAD Document Number 7517

Networking Research in Front Ending
and Intelligent Terminals

UNIX/ENFE EXPERIMENTAL PERFORMANCE REPORT

by

Daniel E. Putnam
Geneva G. Belford
David C. Healy

Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
Defense Communications Agency
Washington, D.C. 20305

under contract
DCA100-76-C-0088

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

September 30, 1977

Approved for Release: Peter A. Alsberg

Peter A. Alsberg, Principal Investigator

Table of Contents

Executive Summary

Background

Under contract DCA100-76-C-0088, the Center for Advanced Computation (CAC) of the University of Illinois at Urbana-Champaign is investigating the capabilities of network front ends. As a part of this contract an experimental network front end (ENFE) was developed to interface a WWMCCS H6000 to the ARPA Network and to conduct experiments with a host-to-front-end protocol. The experimental network front end was developed on a DEC PDP-11/70. The CAC had previously enhanced the UNIX operating system for PDP-11's in order to allow a PDP-11 to act as a mini-host on the ARPA network. Networking software developed by the CAC was further enhanced to support the proposed ARPA Network Host-to-Front-End Protocol (HFP).

The experimental network front end was evaluated in a series of tests and experiments. In CAC Document No. 227, we described a comprehensive set of plans for ENFE tests and experiments. These plans included both the short-term tests to be carried out under the current contract (Phase A) and more extensive testing and analysis to be carried out under a follow-on contract (Phase B). In this document we describe the Phase A tests and summarize the results and conclusions of those tests.

Experimentation Software

The front-end tests used a configuration of software modules similar to the standard ENFE configuration, except that local and foreign host processes were simulated by processes resident in the ENFE itself. These processes served as message generators, sending messages to each other via the standard ENFE modules. An extra copy of the channel protocol module was also included in the ENFE to interface the "local host" message generator to the front-end's channel protocol module.

In order to make accurate timing measurements, a programmable clock was attached to the PDP-11/70. System calls were implemented to enable the experiment software to utilize this clock to get clock readings and to schedule interrupts. Timestamping software was built into the front end at various points. This software inserted clock readings (timestamps) into the texts of messages as they were transmitted through the front end. In this way the progress of a message through the ENFE could be measured to a high degree of accuracy.

Small modifications were made in the standard Unix monitoring facilities to provide for monitoring of kernel-level as well as user-level processes. This allowed a detailed analysis of processor usage.

Experiment Results

A large part of the Phase A testing and evaluation task involved testing the software to make certain that it operates correctly. A certain amount of fine-tuning (more than was anticipated in the Experiment Plan) was also carried out. The Phase A measurements reported here have allowed us to identify which portions of the system need to be made more efficient and to draw broad conclusions regarding the system architecture.

The measurements reported here include:

1. timing tests of the Interprocess Communication (IPC) primitives,
2. timings of the progress of single messages through the front end,
3. monitoring of processor usage, and
4. saturation throughput using several different configurations.

From timing the IPC primitives, we found that it requires a minimum of 5 to 6 milliseconds to relay a message from one front-end process to another. The single-message timing measurements indicate that (except in the case of the NCP) the time a message spends being processed by

the modules is a small fraction of the time required to relay the message between modules. Monitoring the processor usage by the channel protocol module and by the host-host service shows that 85 to 90 percent of CPU time is expended in system calls. Furthermore, kernel-level monitoring shows that about half of this time is just in the overhead of making system calls. We conclude that, as long as the front-end architecture requires Unix system calls to relay messages from one module to another, no drammatic improvement in the efficiency of the front-end services can be expected.

Obtaining meaningful throughput measurements has been made difficult by the self-contained experimentation configuration, which included two passages through the NCP. In this configuration, we found that throughput is severely limited by the NCP. To investigate the extent of this limitation, we have sent messages as fast as possible from the ENFE through the Urbana IMP to an 11/50 at Urbana. With this configuration, each message is handled only once by the ENFE NCP. The maximum throughput measured to date with this configuration is about 50 kilobaud (already enough to saturate the ARPANET) when messages are sent by the 11/50 to the ENFE. However, when messages are sent from the ENFE the throughput is roughly 40 percent less. We attribute this difference to inability of the slower 11/50 to receive data rapidly.

To further investigate the factors affecting throughput, we have separately exercised the two major portions of the message path in the self-contained configuration. The front-end portion of the path (from the message generator to the host-host service) has a message throughput that is four to five times greater than that of the network portion (from a message generator through the NCP to the IMP and back through the NCP to a message receiver). These results provide corroboration of the limiting effect of the NCP.

Introduction

The experimental network front end (ENFE) has been subjected to a program of testing and evaluation in order to:

1. check the correctness of the front-end software,
2. identify system bottlenecks,
3. identify centers of heavy machine resource usage, and
4. estimate the maximum bandwidth.

By identifying system bottlenecks and centers of heavy resource usage, subsequent fine tuning efforts may be able to increase the bandwidth of the front end. Indeed, some improvements have already been made owing to insights gained from the experiments. In addition, subjecting the system to very heavy loads has flushed out bugs that did not surface under ordinary debugging procedures.

The front-end tests use a configuration of software modules similar to the standard ENFE configuration, except that local and foreign host processes are simulated by processes resident in the ENFE itself. Figure 1 depicts the standard configuration of software modules in the ENFE; figure 2 shows the configuration used in the tests. Our tests have concentrated on the path through the host-host service. The program access and server telnet service modules are quite similar in design to the host-host service module. Testing the host-host service module thoroughly exercises the flow control mechanisms and system logic and provides a means for evaluating the basic performance characteristics of this type of front-end architecture.

The leftmost module in figure 2 is the host message generator, which simulates local host processes by generating and receiving messages on one or more channels. The next module is a copy of the standard

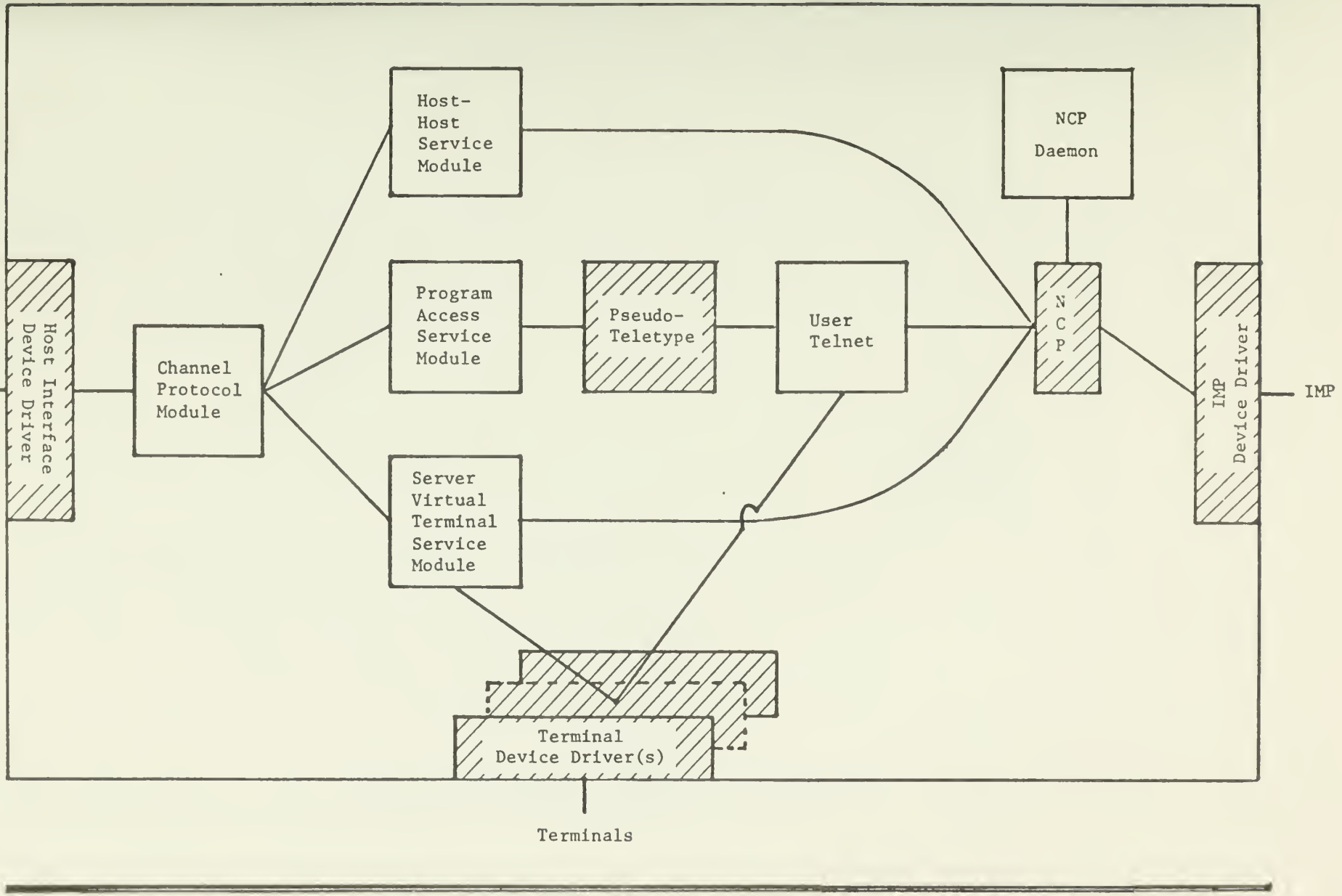
Host-
Host
Service
Module
NCP
Daemon
Program
Access
Service
Module
Pseudo-
Teletype
User
Telnet
N
C
P
H6000
Host Interface
Device Driver
Channel
Protocol
Module
IMP
Device Driver
IMP
Server
Virtual
Terminal
Service
Module
Terminal
Device Driver(s)
Terminals
= software resident in the operating system

Figure 1

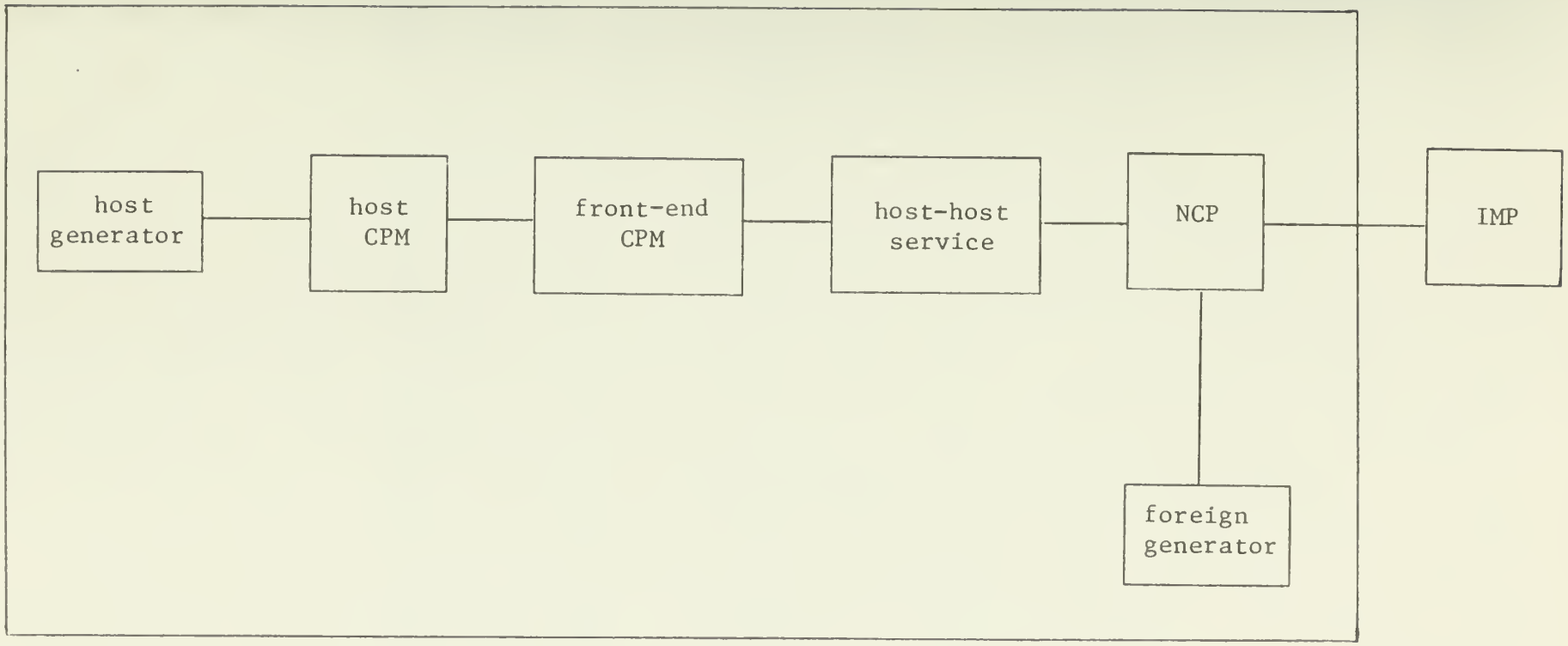

Figure 2

Basic experimental software configuration.

channel protocol module (CPM) and is employed to simulate the local host CPM. Together, the message generator and the host CPM take the place of the local host. The next module is the front-end CPM which communicates with the host CPM just as though it were actually communicating over a hardware link to the real host. The host-host service module and the NCP handle messages to and from the IMP just as they ordinarily would. In the experimentation, messages reaching the IMP are addressed to the ENFE and so the IMP simply returns them. The NCP then delivers them to the foreign host generator module. Traffic also flows in the other direction with messages being generated by the foreign host generator and ultimately being picked up by the host generator.

As messages pass between the two message generators, the messages are timestamped at several points. The amount of time spent at each software module is obtained by examining the record of these timestamps. The total amount of time spent by the various parts of the front-end software may be estimated using an altered version of the standard Unix profiling facility.

In the next section we give a more detailed account of the measurement and evaluation tools used in the experiment program. Then we discuss the details of the software configuration used in the experiments. A final section discusses the results of the experimentation.

Monitoring Facilities

The line clock. Since throughput and processing time are central issues in the operation of a front end, timing has played a key role in measuring the performance of the ENFE. The standard timing method used on a PDP-11 employs the interrupts generated by the cycles in the AC power supply. Unix handles such line interrupts in a routine called "clock" where, among other things, a timing variable is incremented. This variable can be accessed anywhere in the system and is effectively a clock which ticks sixty times per second. Actually, small variations in the line frequency and delays in handling interrupts may cause slight inaccuracy in these readings.

Unix histogram utility. Given a timing mechanism of this sort, the routine which handles the clock (or line) interrupts can collect statistics on which software module was in execution as the interrupt occurred. At each interrupt, the value of the program counter before the interrupt can be examined and a histogram can be built up which estimates the relative amounts of computation performed in each section of a given program. There is a standard Unix system utility called "prof" which enables a user to produce just such a histogram of his own program. To use this facility a user need only invoke the "-p" option on compilation and then interpret the results with the prof program.

We have made some minor changes in the present Unix monitoring system to make it better suited to the needs of the experimental program. For each of the primary user-level modules of the ENFE, two histograms can now be produced. The first monitors the computation performed in the module itself, and the second monitors computation performed in the kernel

at the behest of the given user-level module. By combining the kernel histograms of each of the user-level modules, a picture of the overall activity in the kernel can be obtained as well.

Building this monitoring facility required only a few alterations to the existing system. The routine that handles the line interrupts can now collect histogram data when kernel processes are interrupted as well as when user processes are interrupted. The "prof" program that analyzes the collected data did not need alteration to handle kernel mode as well as user mode. The library routines employed by the compiler when the "-p" option is used have been altered so that they perform the same operations for kernel processes as for user processes. Finally, the "profil" system call has been changed to initiate the dual histogram data collection for a user module. Thus, only minor changes to already existing features of the system have allowed us to obtain a picture of the nature and level of the processing load in the ENFE.

It must be remembered that observations made at regular intervals of one sixtieth of a second are hardly independent. It is conceivable that some loop might take exactly one clock tick to execute. Observations of such a loop would lead to the erroneous impression that one small part of the loop took all the execution time. This effect is probably not important in practice. Since several hundred thousand instructions can be executed each second, there is plenty of room for random effects such as I/O interrupts to intervene between clock ticks spaced at one sixtieth of a second. Therefore we believe that the line clock has been fully adequate for the purpose of carrying out the profiling measurements used in our experiments.

Unix "times" utility. A close relative of the profiling facility of Unix is the "times" utility. Counts are kept of the number of line interrupts which occur during user-level and system-level processing for a given process. The process can obtain these counts by using the times system call. Comparison of these counts is useful in determining the relative amounts of user-level and system-level processing incurred by a process.

The programmable clock. The line clock is clearly not adequate for certain purposes, such as timestamping messages, where much more finely grained and accurate timings are required. For such purposes we have used a hardware device known as a programmable clock. This device is driven by a crystal controlled clock, and its operation can be set by the system. Our programmable clock contains a 32-bit counter which is decremented at a specified rate (100 kHz in our experiments). The value of this counter can be read or reset by the system at any time. When the value of the counter reaches zero, an interrupt is generated and the counter may again be reset. Thus the programmable clock not only provides an accurate means of measuring time, but provides a means of signalling when a given period of time has elapsed.

Two system calls have been implemented to enable the experiment software to use the programmable clock. The first call, "ptime", obtains the amount of time since the system was last brought up. The time interval is measured in units of 10 microseconds. The timestamps placed on messages traveling through the system are the values obtained from calls to ptime. The second call, "pevent", arranges for the operating system to notify a process when a given period of time has elapsed. A process specifies a

length of time (measured in units of 10 microseconds) in a call to pevent. The system then handles the programmable clock so that an interrupt will occur after that length of time. At the interrupt, the system then notifies the process by means of the event mechanism of the interprocess communication facility (IPC) described in [3].

Message Generators

The two message generators shown in figure 2 use ptime and pevent to subject the front-end software to a message load set by the experimenter. For a given experiment the experimenter specifies:

1. the duration of the experiment,
2. the number of open channels,
3. the message sizes for each open channel in each direction,
4. the rate of transmission for each channel in each direction, and
5. the distribution of inter-message transmission time for each open channel in each direction.

The last parameter selects either a constant interval between transmissions or else an exponentially distributed interval between transmissions. The constant mode of transmission is useful in experiments with only one channel open and messages being sent in just one direction at a low rate of speed. This allows the experimenter to measure the processing time required per message by each of the front-end software modules. On the other hand, the exponential mode of transmission allows a more realistic test of the ENFE capabilities. Furthermore, using the exponential distribution injects a richer variety of possibilities into the tests, and may therefore make bugs such as races more likely to surface.

Our present front-end machine is a PDP-11/70 which is not equipped with a hardware floating point processor; therefore it is not feasible for the message generators to compute their own exponentially distributed inter-message transmission times. Even though the 11/70 is a fast machine and there exist all-integer algorithms for computing exponentially distributed numbers, it has proved more efficient to run the message generators from scripts written in advance on disk files. The experimenter specifies the parameters for an experiment as the input to a script-writing program which produces these disk files as its output. A script consists of entries each of which describes a transmission event. Each entry consists of two 16-bit words. One word specifies the interval since the last event; the other word specifies the nature of the present event. The first entry of a script calls for the opening of the specified number of channels. The last entry calls for the closing of all channels and the end of the experiment. The sum of the times given in each script entry is the experiment duration specified by the experimenter.

While the experiment is running, each message generator carries a segment of its script in its own memory space. A third process (not shown in figure 2) replenishes these segments with new event data from the disk as the generators use up their events. As each generator begins its transmissions, it makes a call to ptime to establish a base time for all subsequent transmissions. The intended transmission times for all of the events on a script are then determined by this base time and the inter-event times given in the script entries. A generator may find that the scheduled time of its next transmission event is later than the current time. It then issues a call to pevent so that it will be notified at the intended time of the next event. However, if the generator

is running behind the scheduled event times, it will attempt to send messages until it catches up to its schedule.

It is quite possible for an experimenter to specify a message load that exceeds the bandwidth of the ENFE. In such cases the front-end software routinely resorts to flow control to avoid being overwhelmed by a flood of messages. When a channel is shut off in this way, but the script calls for message transmissions, the first such message is buffered for immediate transmission when the channel reopens. The rest of the messages which can not be sent are simply deleted. This simulates what happens when a real process finds itself unable to transmit. In reality of course no messages are dropped in this way, but statistically the effect of dropping messages with exponential intermessage times is the same as shutting off the message generation process and restarting it when the channel reopens. The great advantage of dropping events that cannot be transmitted rather than shutting off the generator is that the duration of the experiment is not prolonged. The generator moves through the script at just the specified rate.

In addition to sending messages, each generator also picks up the messages that have been sent by the other generator. Again, the absence of a hardware floating point processor precludes the possibility of on-board statistical analysis of the message timestamps. Therefore the timestamps are written out to disk for later processing.

In the course of the experimentation it was found to be difficult to saturate the system using generators of this type. Running off of a script and using ptime and pevent contributes to the computational overhead and therefore limits the speed of message generation. Therefore, two simplified generators were also written in order to provide saturation

loads. These generators write messages as fast as flow control will allow. Also, no timestamping or recovery of timestamps is performed. The object is simply to load the system as fully as possible.

Other Front-end Modules

Host CPM. The host message generator communicates with the "host CPM" module shown in figure 2. The normal path between a process in the local host and a service in the front end takes messages from the local host process to the local host CPM, over a hardware link to the front-end CPM, and then to the front-end service. To simulate this situation we have interposed a copy of the standard front-end CPM between the host message generator and the front-end CPM. This CPM copy takes the place of the CPM in the local host. Thus we have used two separate modules to simulate the local host CPM and the local host processes, rather than just one module to simulate the whole local host. This has allowed an additional check of the correctness of the CPM and of the channel protocol itself.

Standard modules. The message path between the host CPM and the foreign message generator is the same as in the standard ENFE. The front end CPM, the host-host service module, the NCP and the IMP stand in their usual configuration and perform their standard functions. Thus the main portion of the message path through the standard ENFE has been left intact for the purpose of these experiments. The only part of the standard path which has been omitted is the module which handles the hardware link to the local host machine.

Message Transmission

Messages traveling along the path between the host message generator and the host-host service module are in the form of HFP Commands

or Responses [2] and are passed between modules via the IPC message mechanism [3]. For example, the host message generator opens a channel by sending an HFP BEGIN Command to the host CPM module. Later, the host CPM will relay a BEGIN Response back to the host message generator and the channel will be open for transmission. HFP TRANSMIT Commands are used as described in [2] to send message data from the host message generator to the host-host service module for transmission to the foreign host module via the IMP.

Data passing between the host-host module and the NCP, or between the foreign host generator and the NCP, is transmitted via system read and write calls, not the IPC message mechanism. Such data is effectively broken down into a stream of bytes which the NCP re-packages for transmission to the network. The packets sent by the NCP to the IMP are addressed back to the ENFE; therefore the IMP returns them as part of its normal function. The NCP passes the returning data as a stream to the host-host module or the foreign host generator, depending on the direction of transmission.

Timestamping

One strategy for studying front-end performance has been to timestamp messages passing between the two generators. Since there is no need to transmit actual message data, the timestamps are simply written in the texts of the messages themselves. The resulting record of clock readings marks the progress of a packet through the various front-end modules. In this way the component parts of the processing time for a packet may be individually measured and studied.

To conduct this type of analysis it is necessary that messages maintain their identity as single packets. Each timestamping location in the front end is assigned an offset from the beginning of a message;

i.e., a unique field in the message text where the timestamp is to be placed. If messages are split up or merged it becomes infeasible to timestamp in such a way that the results can be sorted out and studied.

On the path between the host message generator and the host-host service module, message packets are handled as distinct entities. No TRANSMIT Command is ever split into two or more smaller Commands nor are two or more distinct TRANSMIT Commands ever combined into a larger Command. Therefore, packets sent by the host generator can be time-stamped to mark their progress up to the point where they enter the NCP. Unfortunately, the packets lose their identities as they enter the NCP, so that no timestamping can be done inside the NCP-IMP portion of the message path. When the NCP eventually passes message data to the foreign generator, the foreign generator can reassemble the byte stream into the original packets that were sent by the host generator. In fact, the leading byte of each of these packets indicates the length of the packet. Therefore, the foreign generator can compute the location of packet boundaries in the byte stream passed from the NCP. When the foreign generator reconstitutes one of the original packets, it recovers the timestamps written up to the point where the packet entered the NCP. Then it adds an additional timestamp to mark the arrival of the packet.

Thus, messages sent from the host generator to the foreign generator can be timestamped at multiple locations up to the point where they enter the NCP and once when they arrive at the foreign generator. Timestamps are applied at the following points:

1. When the host generator sends the packet to the host CPM.
2. When the host CPM receives the packet.
3. When the host CPM sends the packet to the front-end CPM.
4. When the front-end CPM receives the packet.
5. When the front-end CPM sends the packet to the host-host module.

6. When the host-host module receives the packet.

7. When the host-host module enqueues the packet for transmission to the NCP.

8. When the host-host module sends the packet to the NCP.

9. When the packet arrives at the foreign generator.

It is not feasible to timestamp messages sent from the foreign message generator to the host generator in the same manner as traffic in the opposite direction is timestamped. When the foreign generator attempts to send message packets, the packets lose their identity as soon as they are sent. Transmissions from the foreign generator are immediately reduced to a stream of bytes by the NCP. An attempt to send a packet may result in part of the packet being immediately taken by the NCP while the rest of the packet must wait until the NCP is ready to handle more data. The byte stream emerges from the NCP in the form of packets passed to the host-host service module, but these packets bear little resemblance to the original packets. Of course the bytes are all in the right order, but a packet which reaches the host-host module may consist of fractional parts of one or more of the original packets sent by the foreign message generator.

For this reason a simpler variety of timestamping is applied to messages sent from the foreign side of the front-end to the host side. The foreign generator prepares message packets and timestamps the tail end of the packet just before the initial attempt at transmission. Furthermore, the total length of the packet is written in the first byte of the packet. Using this data, the host message generator can reassemble the original packet when all of its bytes arrive and recover the timestamp. Thus the total time through the system in this direction can be obtained.

It was not expected that transmission through the front end would be any faster in any of the modules in one direction than in the other. This timestamping strategy permits a check of that hypothesis by comparison with transmission times from the host generator to the foreign generator. This comparison is discussed in the results section below.

Experiment Results

A large part of the contract effort allocated to tests and experiments was necessarily taken up in testing the software to make certain that it operates correctly. A certain amount of fine-tuning was also carried out. For example, when the simple XON/XOFF flow control mechanism (see [4]) was discovered to be unsatisfactory, a credit strategy was designed and implemented. This necessitated changes in the experiment software, etc. As another example, it developed that under heavy message loads, the IPC mechanism [3] could lock up for lack of free segments. It had been known that this was a possible difficulty, but until we ran the tests it was not clear whether this condition would actually occur. Segment swapping has been implemented to avoid this problem and is currently undergoing tests. Thus, instead of expending our efforts solely on system testing, we have pursued a program of concurrent testing and tuning. Indeed this process is continuing at this writing.

The measurements reported here include timing tests of the IPC primitives, timing the progress of single messages through the front end, and saturation throughput and profiling.

Timing the IPC Primitives

Because the IPC mechanism [3] provides the basis for communication between many of the front end modules, the IPC primitives were the first part of the system to be tested. Measurements were taken of the time to execute the various interprocess communication primitives. The measurements were carried out by repeatedly executing program segments of the form:

ptime

IPC primitive

ptime

Differences between the two times returned by ptime were accumulated and averaged. Certain adjustments were made in the raw measurements to arrive at the numbers which we report here as "adjusted mean execution times."

First, when a particularly long interval (specifically, twice as long as the running average up to that point) was measured, it was not included in the computation of the mean. The rationale for dropping such measurements was that the long delay was almost certainly caused by loss of the processor and should not be included in the primitive execution time. A count was kept of the number of such measurements that were dropped to make sure that they did not occur too frequently.

Second, since ptime itself takes up some time, measurements were also taken for

ptime

ptime

This test overhead time was subtracted from the average execution times computed for the primitives.

Four separate experiments were carried out. In each one, each primitive was executed 4000 times. The experiments were done at different times of day to compare timings in a "busy" machine with those in an "empty" machine. The adjusted mean execution times of most primitives were roughly 5 to 7 percent longer in a busy machine. Some primitives--specifically getsba and getseg-- showed virtually no increase in execution time during busy periods. A more dramatic effect of running in a busy machine was that far more measurements were discarded for being too long. That is,

processor loss becomes a factor in IPC efficiency, occurring from 1 to 3 percent of the time. (In an "empty" machine, about 0.1 percent of the measurements were discarded.)

The adjusted mean execution times for the four separate experiments were averaged together to get the results presented in table 1. We did not compute variances, since close examination of the distribution of the measurements indicated that that statistic had little meaning. A typical distribution featured one very large, sharp peak, containing about 90 percent of the measurements within a time range of about 50 microseconds. Clearly, the basic execution time was to be found within that range. The rest of the distribution was skewed towards longer measurements--in fact, no measurements were below the large peak. A second distinct, but relatively small, peak appeared about 0.2 ms after the first peak. We ascribe this to the occurrence of some common interrupt during execution. Longer times can probably be ascribed to other interrupt handling and, in extreme cases, loss of the CPU.

Total time to relay a message. To send a message, a process must execute the following sequence of primitives:

```
getseg
mapseg (sba, segname)
sndseg
freeseg
```

The time to execute these primitives represents a minimum time to send a message. Interrupts and process swapping will add to the time. In addition, if a message is not just being relayed, the process will spend some time building the message. From table 1, we see that the mean total execution time for the set of five primitives listed above is 3 ms.

IPC Primitive	Time
Sndevent	0.74
+Wevent	0.87
+Wall	0.86
Rall	0.87
Getsba	0.35
Freesba	0.47
Getseg	0.81
Freeseg	0.72
Mapseg (sba, segname)	0.62
Mapseg(sba,-1)	0.50
*Sndseg	0.82

Table 1

Adusted mean execution times (in milliseconds)

averaged over four experiments of 4000 executions each.

* Sndseg was included in only two of the four experiments.

+ To avoid inclusion of waiting time in the measurements of wevent and wall, an event was put into the event queue before the timing code was executed.

To receive a message, a process must execute the following sequence of primitives:

wall

mapseg

freeseg

(Freeseg is not called until after the message is processed, but it must be included here as a necessary part of receiving a message.) The mean total execution time for this set of four primitives is 2.2 ms. Adding the receiving time to the sending time, we conclude that the total time for one process to send a message and a second to receive that message is 5 or 6 ms.

Timing of Single Messages

The timestamping message generators were used to measure the processing time required by a single message in an empty machine. In these tests the host generator sent TRANSMIT Commands to the foreign generator at intervals of one second. This is much more than enough time for a message to clear the system before the next message is sent. The differences between timestamps record the amount of time spent in each part of the system. Tables 2a and 2b show the results of two such tests with 300 sample points each. It is difficult to prevent the contamination of test results by outside events such as an attempt by another host to open a connection to the ENFE. For this reason we eliminated one message from the test run of table 2a. The message in question took more than four times as long as the other messages to traverse the system. The rows of the tables represent the following time intervals:

1. from the host generator to the host CPM,
2. inside the host CPM,
3. from the host CPM to the front-end CPM,

4. inside the front-end CPM,

5. from the front-end CPM to the host-host service,

6. inside the host-host service,

7. from the host CPM to the NCP,

8. from the host generator to the foreign generator.

The first of these tests used small messages consisting of 55 bytes each. Along with the nine-byte header in an HFP TRANSMIT Command, such a message uses up one 64-byte IPC segment. The second test used 439-byte messages, enough to fill 7 IPC segments. This is just slightly less than the agreed upon maximum message size of 3600 bits.

In lines 1, 3, and 5 of the tables we see approximately 3 milliseconds for transmission between adjacent modules via the IPC mechanism. These mean times are derived from the difference of timestamps taken before the sendseg done by the sending process and after the wall done by the receiving process. In lines 2, 4, and 6 are the data for message handling times by the host CPM, the front-end CPM and the host-host service. Here the mean times differ greatly. The host CPM and the host-host service modules require only about 1.5 milliseconds per message. The front-end CPM takes 5 or 6 milliseconds. The front-end CPM and host CPM are identical programs, but they are performing different tasks in these tests. The extra time taken by the front-end CPM is due to sending TRANSMIT Responses back to the host CPM. This uses five additional IPC calls:

a getseg,

two mapsegs,

a sendseg and

a freeseg.

Path Segment	Mean	Standard Deviation	Minimum	Maximum
1	2.98	.10	2.87	3.16
2	1.71	.006	1.70	1.73
3	3.13	.05	3.10	3.60
4	5.65	.11	5.56	5.86
5	3.11	.11	3.00	3.26
6	1.35	.10	1.27	1.78
7	13.23	.01	13.02	13.73
8	57.07	8.33	46.80	77.65

Table 2a.

Timing measurements in milliseconds for a single-message experiment with small message size (55 bytes). The first column identifies the portion of the path time . (See text for description.)

Path Segment	Mean	Standard Deviation	Minimum	Maximum
1	2.99	.63	2.88	12.09
2	1.76	.08	1.72	2.17
3	3.13	.11	3.06	3.55
4	5.85	.15	5.74	6.74
5	3.08	.20	3.00	6.02
6	1.28	.07	1.25	1.75
7	13.34	.23	13.10	16.43
8	156.26	8.89	61.81	182.79

Table 2b.

Timing measurements in milliseconds for a single-message experiment with large message size (439 bytes). The first column identifies the portion of the path timed. (See text for description.)

Sending back a TRANSMIT Response for every TRANSMIT Command received adds a significant load to this part of the system. Two kinds of improvements are possible. The overall strategy for sending back TRANSMIT Responses can be improved to allow fewer Responses to be sent. Furthermore, the code in the CPM can be altered so that only one of the two mapsegs is required. At this writing both of these improvements are being implemented.

The seventh line of the tables represents the amount of time spent by a message from the time it is sent to the front-end CPM until it is passed to the NCP. These mean times are substantially smaller than the total transit times shown in the last row of the tables. The NCP-IMP portion of the path clearly takes the largest share of the total time. For small messages, the time in the NCP-IMP is about 40 milliseconds; for large messages it is about 140 milliseconds. Large messages take longer on the network portion of the path because messages are broken down into a byte stream as they enter and leave the NCP. It must be remembered that the NCP plays a dual role in these experiments. It not only performs its usual function in sending messages out to the IMP, but it also simulates the NCP of a foreign host when it handles messages coming back from the IMP. More will be said later concerning the extent to which the NCP limits system throughput. However, it seems clear that tuning the NCP could result in a considerable increase in performance.

Single message traffic was also sent in the opposite direction, from the foreign side to the host side, to compare total transit times. Recall that complete timestamping is not feasible for traffic in this direction since messages lose their identity upon entering the NCP. However, total transit times can be obtained. For small message traffic

this time is about 56 milliseconds, a time fully comparable to that of small messages sent in the other direction. This is just what we had expected. However, when large messages are sent, the transit time increases to about 170 milliseconds. Occasionally, the NCP breaks up large packets into smaller packets, which are handled as separate TRANSMIT Commands. When traffic is sent in the other direction, from host side to foreign side, large messages are handled in a single TRANSMIT Command. Therefore less processing is required.

Saturation Tests

Many more tests were run using the timestamping apparatus. These tests were run under a variety of conditions. However, when the timestamping generators are used to try to saturate the system, the results of analyzing the timestamp records are less interesting. Indeed the mean times for message transmission between front-end modules or for processing by the modules are measured in hundreds or even thousands of milliseconds. These times simply reflect the facts that a message routinely sits in IPC queues for long periods of time and that a module will routinely lose the processor while handling a message. Thus, under heavy loads it becomes more fruitful to measure such quantities as throughput and processor usage. The simplified generators which write as fast as allowed by flow control are better suited to these purposes. Because the simplified generators avoid the overhead of timestamping and running off of a script, they can impose greater loads on the system. Two series of tests were run with these generators to simulate message traffic sent from the host side of the front end to the foreign side and from the foreign side to the host side. Tests with traffic flowing in only one direction simulate file transfers to and from the host. The rate of throughput expressed in kilobauds (1000 bits per second) was measured by dividing the quantity of data sent during the test by the length of the test.

Table 3a shows the observed throughput rates for traffic sent from the host side to the foreign side. Table 3b shows throughput rates for traffic flowing in the opposite direction.

At this point it must be recalled that this experimental configuration contains as much or more experiment software as standard software. The host generator, the host CPM and the foreign generator are not part of the standard system. In addition, the NCP is doing the work of both a front-end NCP and a foreign NCP. Precise measurements of the throughput of the standard ENFE will have to wait until the ENFE is connected to a host and used with its standard software configuration. Installing experiment software in the ENFE along with the standard software has been tremendously useful in flushing out bugs and in testing logical correctness. However, the presence of experiment software in the ENFE must inevitably degrade system performance.

We attempted to derive better estimates of the throughput of the standard ENFE by using some different test configurations. First, we moved the foreign generator to the 11/70 at Reston and sent messages to and from that machine via the ARPANET. This was easy to do since that machine runs the same ENFE system as the 11/70 in Urbana. This configuration has the advantages that the Urbana system must no longer support an extra message generator and that half of the NCP burden is shifted to the 11/70 at Reston. However, we found that throughput dropped by at least a factor of two. Obviously the throughput potential of one machine is less than the potential of two machines which share the processing load. The problem here is that the ARPANET was simply unable to run fast enough to push the systems to their limits. ARPANET traffic is slowed by flow control between the IMPS which relay message traffic between hosts. The slowness of the network in this case is explained by the presence of five IMPS between Urbana and Reston.

Number of Channels	Combined Throughput
1	21 kilobaud
2	23 kilobaud
4	27 kilobaud

Table 3a

Throughput observed in transmission from host side to foreign side.

Number of Channels	Combined Throughput
1	22 kilobaud
2	25 kilobaud
4	28 kilobaud

Table 3b

Throughput observed in transmission from foreign side to host side.

To avoid this problem, we obtained dedicated system time on a PDP-11/50 which is connected to the same IMP as the ENFE. We repeated the experiments described above with the foreign generator resident in the 11/50. Tables 3c and 3d show the throughput rates observed in message traffic to and from the 11/50.

Two facts are important in interpreting these results. First of all, the computational load on the NCP is substantially larger when receiving messages than when sending them. Furthermore the 11/70 is much faster than the 11/50, perhaps by as much as a factor of two. In the tests of table 3c, the NCP of the slower 11/50 must bear the load of receiving messages while the 11/70 sends them. When these roles are reversed in the tests of table 3d, we see throughput increase substantially. Apparently the 11/50 is the bottleneck in the tests of table 3c. However, it is not clear whether the 11/50 or the IMP or the 11/70 is the bottleneck in the tests of table 3d.

The primary information which we wish to convey here is that the throughput seen in our earlier tests, (tables 3a and 3b), was very severely limited by the double load on the NCP along with the load of the foreign generator. The foreign generator does very little else besides system reads and writes to communicate with the NCP. Evidently the NCP, along with the mechanism for communicating with the NCP, is a very important factor in limiting throughput. When we eliminated the smaller sending load on the 11/70 NCP in the table 3d tests, throughput increased by about a factor of two. The front-end system tested in these two-machine experiments is still burdened by the presence of experimental software. However the throughput observed in these tests is a much better estimate of the real capabilities of the ENFE.

Number of Channels	Combined Throughput
1	28 kilobaud
2	30 kilobaud
4	32 kilobaud

Table 3c

Throughput observed in transmission
from ENFE to 11/50.

Number of Channels	Combined Throughput
1	39 kilobaud
2	52 kilobaud
4	51 kilobaud

Table 3d

Throughput observed in transmission
from 11/50 to ENFE.

At this point we turn our attention back to tests in which all software resides in the Urbana 11/70 machine. Two additional test configurations were employed in an attempt to identify which part of the message path was most important in limiting message throughput. We may distinguish here between the "HFP" portion of the message path and the network portion of the path. By the HFP portion we mean that part of the path between the host generator and the host-host service module. Communication between these modules takes the form of HFP Commands sent via the IPC mechanism. By the network portion of the message path we mean the path into the NCP, out to the IMP and back into the NCP. With a little extra coding we were able to separately exercise each of these parts of the message path.

To exercise the HFP part of the path, a special test service was written to take the place of the host-host service module. This test service can be operated as a message sink, receiving TRANSMIT Commands from the front-and CPM just as the host-host service would in its place. However, it does not pass message data along to the NCP via system write calls; it simply drops the data. Alternatively, the test service can be operated as a message source. In this mode of operation it sends TRANSMIT Commands to the front-end CPM as fast as flow control permits. Whereas the host-host service module must obtain data from the NCP via system read calls, the test service is always ready to send data.

Several tests were performed which used the test service either as a source or as a sink (but not both) and used a varying number of open channels. The throughput observed in these tests was remarkably stable regardless of the direction of traffic flow or number of channels. In each case rates of approximately 130 kilobaud (36 3600-bit messages per second) were observed. This is 4 or 5 times the rates observed on the full experimental path between host generator and foreign generator when all software resides in the ENFE. Clearly the network portion of the message path severely limits system throughput.

For the sake of comparison with the above tests, additional tests were performed in which the HFP portion of the message path was eliminated. In these tests two copies of the foreign generator communicate with each other via the NCP and IMP. One copy of the foreign generator acts as a message source, writing as fast as the NCP can handle data. The other copy of the foreign generator acts as a message sink and only serves to collect data from the NCP. Like the host-host module, the receiving generator obtains its data from the NCP via system read calls. However, it omits the host-host function of passing this data along to the front-end CPM in HFP TRANSMIT Commands. Here again we expected to see an increase in throughput over that observed on the full message path. But the increase is rather small, the observed rates of throughput being only about 30 kilobaud, depending on the number of open channels. Thus, there is very little gain in eliminating the HFP portion of the path.

Profiling

Besides studying throughput we have also measured processor usage by the front-end software modules. The results presented here were obtained from the same saturation tests that produced the data in tables

3a and 3b. The simplified high speed generators were used to send messages in one direction or the other on varying numbers of channels. Processor usage is remarkably stable when a given test is rerun for comparison. But even when the direction of message flow is reversed or the number of channels is changed, processor usage generally varies within only a small range. Table 4 presents the percentage of total processor time used by the software modules in these tests. The total fraction of processor usage by these modules is always in the 40 to 60 percent range. It is difficult to pin down the remaining time except to say that it may be attributed to the network portion of the message path. This time is divided between idle time, when the system is waiting for the IMP, and actual processing done by the NCP.

It is clear from table 4 that the modules in the HFP portion of the message path take only a moderate amount of processor time. In view of this fact it is especially interesting to note that the greatest share of this time is system-level processing. Using the "times" utility we have observed that system processing time routinely represents 85% of the processing time taken by the CPM modules. For the host-host service module the corresponding fraction is even larger, often exceeding 90%. Virtually all the system overhead incurred by these modules originates in system calls used to transfer message data to another module. For the CPM modules these system calls are the IPC calls. The host-host service module uses these calls to communicate with the front-end CPM, but it also uses system read and write calls to communicate with the NCP. System read and write calls are more expensive than IPC calls because message data is actually moved byte by byte. This extra expense is reflected in the higher level of system processing expended by the host-host service module.

Module	Percent usage
host generator	2-6
host CPM	5-9
front end CPM	5-10
host-host service	8-15
foreign generator	11-20

Table 4

Range of percentage of total processor time used by the software modules.

It becomes clear at this point that no major improvements can be made in the performance of the front end by trying to decrease user-level processing in the CPM or host-host service modules. As we noted earlier, the CPM is being improved by decreasing the number of IPC calls that it must make. Possibly, something similar can be done with the host-host service module. Aside from decreasing the use of system calls to communicate between modules, any further improvement must come from refinement of the system calls themselves.

With this idea in view, we used the kernel-level profiling apparatus on the CPM and host-host service modules to see which parts of the system calls were taking the most time. Kernel-level profiling yields a histogram indicating which subroutines were found to be in execution at the arrival of line interrupts. The usual goal of such an analysis is to order the routines by decreasing usage. However, determining one consistent ordering proved to be impossible in this case. About 10 or 15 routines generally account for 85% of the system time. The time is fairly evenly spread among these routines. In fact the most heavily used routine usually accounts for no more than 15% of the total time. Because of the even time distribution, a consistent ordering is not observed from one test to another, even when the tests are fairly long. Thus it is inappropriate to present an ordered list of the most heavily used routines. However we can list several of these routines with the expectation that in a given profiling analysis they will generally be among the most heavily used and take up a large share of the processing time.

Kernel profiling of the CPM and host-host service module yield very similar results. Virtually all system time taken by these processes results from system calls used to communicate message data between modules. Both the CPM and host-host service make heavy use of the IPC

mechanism for this purpose. Of course, the host-host service uses non-blocking I/O to communicate with the NCP, but non-blocking I/O utilizes the IPC event mechanism.

The following appear to be the most heavily used routines.

"suword", "trap" "fuword", "call" "csv" and "cret"	These routines are associated with the general overhead of system calls. The first two usually take 20 to 25% of the time. Altogether they consistently take about 50% of the time.
"ipcrcv" and "wakeup"	These routines are used when a process uses an IPC wall or rall to pick up data from its IPC queue.
"ipcldreg" and "sureg"	These routines set up the relocation registers when a process switch occurs.
"ipctrans" and "ipcsnd"	These routines are used when a process sends data via the IPC mechanism.

Conclusions

A major goal in taking the measurements discussed in this document has been to identify portions of the system where improvements might substantially increase performance. Two areas of the system stand out as candidates for improvement. The best candidate is the NCP, which seems to severely limit system throughput. The second candidate is the mechanism used to communicate message data between modules. This is not to say that these parts of the system are inefficient. It is only that substantial amounts of system resources go to these functions. Other areas of the ENFE seem to take so little time that improvements would have only a tiny effect on system performance as a whole.

The NCP is a large section of code which performs a number of complicated functions. Given that these functions must be performed there is a limit to the amount of improvement that is possible. However, it should be worthwhile to expend some effort in tuning the NCP to see just how much improvement can be obtained.

In the case of the IPC mechanism it is less clear that tuning might be worthwhile. Given the architecture of the ENFE, which requires that messages be passed between modules via system calls, the factor limiting IPC efficiency appears to be the overhead in making the system calls themselves. High overhead of system calls is a basic property of Unix. A front-end based on the Unix system cannot avoid this overhead.

References

1. Belford, G.G. and Putnam, D.E. "Experimental Network Front End Experiment Plan", CAC Document Number 227, Center for Advanced Computation, University of Illinois at Urbana-Champaign, 1977.

2. Grossman, G.R. "ARPANET Host-Host Process-to-Service Protocol Specification", CAC Technical Memorandum Number 80, Center for Advanced Computation, University of Illinois at Urbana-Champaign, 1977.

3. Holmgren, S.F.; Healy, D.C.; Jones, P.B.; and Kasprzycki, E. "Illinois Interprocess Communication Facility for Unix", CAC Technical Memorandum Number 84, Center for Advanced Computation, University of Illinois at Urbana-Champaign, 1977.

4. Holmgren, S.F.; Kasprzycki, E.; Healy, D.C.; and Jones, P.B. "Experimental Network Front End Software Functional Description", CAC Document Number 233, Center for Advanced Computation, University of Illinois at Urbana-Champaign, 1977.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
CAC Document Number 239
CCTC-WAD Document Number 7517

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Network Research in Front Ending and Intelligent Terminals - UNIX/ENFE Experimental Performance Report

5. TYPE OF REPORT & PERIOD COVERED
Research

6. PERFORMING ORG. REPORT NUMBER
CAC #239

7. AUTHOR(s)
Daniel E. Putnam
Geneva G. Belford
David C. Healy

8. CONTRACT OR GRANT NUMBER(s)
DCA100-76-C-0088

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Command and Control Technical Center
WWMCCS ADP Directorate, 11440 Isaac Newton Sq.,N.
Reston, Virginia 22090

12. REPORT DATE
September 30, 1977

13. NUMBER OF PAGES
44 pages

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Copies may be requested from the address given in (11) above

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restriction on distribution.

18. SUPPLEMENTARY NOTES

NONE

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Network front end
Computer system measurement

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The CAC is engaged in an investigation of the benefits to be gained by employing a network front end. A DEC PDP-11/70 is being used as front end for connecting a Honeywell 6000 host to the ARPANET. This document presents the results of a program of testing and experimentation carried out with this experimental front end.

FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)